Maksym Stepanenko



Technical Debt: How a Startup Kills Itself with Its Own Code

What tech debt is, how to measure it, and when to pay it down

Maksym Stepanenko · 2026

Technical Debt: How a Startup Kills Itself with Its Own Code

What tech debt is, how to measure it, and when to pay it down

Maksym Stepanenko

Innova Publishing · 2026

Book page, latest version and the other guides in the series:
guides.mmix.biz/en/tech-debt

CHAPTER 1

What Technical Debt Is and Where It Comes From

Technical debt is a metaphor proposed by Ward Cunningham in 1992. A quick solution now means taking out a loan from the future. This is not necessarily bad, and sometimes it is the strategically right call. The problem begins when the debt accumulates and starts slowing down development, charging “interest payments” in the form of slow delivery and a large number of bugs.

Where tech debt comes from. Time pressure (the MVP rush): “We need to launch in 2 weeks.” Decisions are made fast, tests are not written, and the architecture is not thought through. This is normal and often right at the pre-product stage. A growing team: different developers have different standards. Without a code review culture, everyone writes in their own way, and inconsistency accumulates. Changing requirements: the architecture was designed for one use case. Requirements change, and hacks and workarounds get added instead of a redesign. Knowledge silos: “only Vasya knows how the authorization module works.” Vasya quits. Nobody understands the code. Outsourced development: external contractors optimize for completion, not maintainability. Goodbye, architecture.

Types of tech debt according to Martin Fowler's quadrant (the Technical Debt Quadrant, martinfowler.com, 2009; the debt metaphor itself was proposed by Ward Cunningham). Reckless deliberate: “We have no time to think about architecture, we're launching.” Prudent deliberate: “We know it's not ideal, but we need to ship now, and we'll fix it later.” Reckless inadvertent: “We didn't know we were doing it wrong.” Prudent inadvertent: “Now we understand how we should have done it.” Reckless deliberate is the most toxic. Prudent deliberate is normal business practice when managed properly.

When tech debt kills startups: the startup grows. It hires more developers. Velocity falls (it does not rise!) with every new hire. Every new feature requires touching 10 places in the code. One bug fix creates three new ones. The CEO: “We hired 5 more developers, so why are we deploying slower?” This is advanced-stage tech debt poisoning. Signs: the cycle time of a feature from idea to production is > 4 weeks. More than 30% of the team's time goes to bug fixes and incidents. Developers are afraid to touch certain parts of the code. Onboarding a new developer takes 2+ months.

CHAPTER 2

How to Measure Tech Debt Objectively

"We have a lot of tech debt" is a subjective feeling. The CEO will not buy it without data. Here is how to measure it objectively.

DORA Metrics (DevOps Research and Assessment): four metrics that correlate with business performance. Deployment Frequency: how often do you deploy to production? Elite: multiple times per day. Low: once per month or less. Lead Time for Changes: from commit to production. Elite: < 1 hour. Low: 1–6 months. Change Failure Rate: the % of deployments that cause an incident. Elite: 0–15%. Low: 46–60%. Time to Restore Service: how quickly you recover after an incident. Elite: < 1 hour. Low: 1–6 months. High tech debt means poor DORA metrics. This is objective evidence for the CEO.

Cycle Time: from "start" (a developer picks up a ticket) to "done" (the feature is in production). Measure it in Jira, Linear, or your project management tool. The trend matters more than the absolute number. If cycle time grows with the same team, tech debt is accumulating.

Defect Rate: the % of bugs reported per feature shipped. If it grows, quality is falling. Code Coverage: the % of code covered by automated tests. It is not a golden metric (70% coverage can be useless if the tests are bad). But a complete absence of tests is a risk indicator.

Developer Experience Survey: once a quarter, ask the team: "On a scale of 1–10, how easy was it for you to make changes in the last 2 weeks?" "What got in the way of your work the most?" "Are there parts of the code you are afraid to touch?" Qualitative data plus quantitative data gives the full picture.

SonarQube and CodeClimate: static code analysis tools. They count code duplication (DRY violations), complexity (cyclomatic complexity), code smells (patterns associated with problems), and

Technical Debt in hours (an estimate of how long it would take to fix). SonarQube: free, self-hosted. CodeClimate: $50+/month for teams.

CHAPTER 3

Prioritization: When to Pay Down Debt and When to Ignore It

Not all tech debt is worth paying back. The main question: which debt slows growth the most?

The Tech Debt ROI framework. For each debt item, estimate: Impact on velocity (1–5): how much does this debt slow delivery? Impact on reliability (1–5): how often does this debt cause bugs or incidents? Effort to fix (1–5, where 5 = a lot of effort). ROI score = (Velocity + Reliability) / Effort. Prioritize in descending order of ROI score.

Tech debt categories by urgency. P0—Fix Now: it causes production incidents. Security vulnerabilities. It blocks key customer requests. P1—Fix This Sprint: it significantly slows development. It is often discussed in standups as friction. P2—Schedule for Next Quarter: it reduces code readability. It will slow future features. P3—Track but Don't Fix Yet: a code smell with no practical impact. It may never be touched.

The “Boy Scout Rule” (gradual improvement): every time you touch a piece of code, leave it a little better than you found it. Do not refactor everything. Just add a test for the function you changed. Improve the name of a variable that confused you. Delete dead code you come across. This is organic debt reduction without special sprints.

20% time: set aside 20% of developer time for tech debt every sprint. This is not a “nice to have.” It is an investment in future velocity. Frame it for the CEO: “If we don't invest 20% now, next year we will be doing the same work 2x slower.” Debt Sprints: once a quarter, a full sprint devoted only to tech debt. This works when the debt has piled up to a critical level. The downside: developers lose context on the product.

Never do this: a full rewrite “from scratch.” A Big Bang Rewrite is almost always a failing strategy. It takes 3–5x longer than expected. The product stands still while the rewrite goes on. The “new” code accumulates its own debt. The alternative: the Strangler Fig pattern, a gradual replacement piece by piece.

CHAPTER 4

How to Talk About Tech Debt with the CEO and Investors

Technical debt is invisible to non-technical stakeholders. The CTO's or Tech Lead's job is to make it visible through business language.

Translate technical metrics into business metrics. Instead of “we have high cyclomatic complexity” → “every new feature takes 2x more time because of the complexity of the code.” Instead of “we have no unit tests” → “we find 40% of bugs in production instead of in development, which costs us N days of recovery.” Instead of “tech debt” → “engineering debt” or “infrastructure investment.” Words matter.

The business case for tech debt investment. ROI calculation: current velocity is 5 features/month. At the current rate of debt growth, in 12 months velocity will drop to 2 features/month. Developer cost: $50,000/month. With 50% of engineering output going to bug fixes instead of new features, that is $25,000/month in “losses.” An investment of 2 months of debt reduction costs $100,000. Result: velocity returns to 5+ features/month. ROI: it pays back in 4 months.

Visualizing tech debt: a burn-up chart with a velocity trend. It shows velocity falling over time. Bug rate over time. Incident frequency. These charts are clear to any CEO.

Pre-funding technical due diligence: investors and acquirers do technical due diligence. Red flags: a monolith without documentation. No tests. No CI/CD. Security vulnerabilities (they will find them with SAST tools). Key-person dependency (only one person can read the code). Prepare in advance: basic CI/CD. Documentation of key architectural decisions. Secret scanning in the repository. A dependency audit (outdated libraries with known CVEs).

A culture of quality: tech debt is a symptom, not a cause. The cause is a culture that does not value code quality. How to create a culture of quality: code reviews are mandatory (even if slow). The Definition of Done includes tests. A “broken window” policy: you do not leave broken code behind. Blameless postmortems after incidents. Celebrate refactoring as much as new features.

CHAPTER 5

Tools and Practices for Managing Debt

The right tools make tech debt management systematic rather than reactive.

CI/CD as the foundation. Continuous Integration: an automatic build and tests on every PR. GitHub Actions, GitLab CI, and CircleCI are free for small teams. Continuous Delivery: an automatic deploy to staging after merge. Continuous Deployment: an automatic deploy to production. Without CI/CD: manual, slow, error-prone. With CI/CD: fast, reliable, and it removes the human factor from deployment.

Automated tests. Unit tests test individual functions. They are fast. Integration tests test the interaction of components. E2E tests test user flows in the browser (Playwright, Cypress). Do not chase 100% coverage. Start with: tests for business-critical flows. Tests for places that break often. Tests for every new bug (regression tests).

Monitoring and observability: Datadog, New Relic, or self-hosted Grafana plus Prometheus. Core metrics: application performance (p95, p99 latency). Error rate. Database query performance. Infrastructure health. Sentry or Bugsnag: automatic error tracking. Every unhandled exception becomes a ticket in your tracker. PagerDuty or OpsGenie: incident management and on-call rotation.

Documentation. Architecture Decision Records (ADR): short documents about key architectural decisions. Why did we choose PostgreSQL over MongoDB? Why microservices instead of a monolith? This is priceless 12 months later. A README.md in every repository: how to run it locally. How to run the tests. How to deploy. An onboarding guide for new developers.

Dependency management: Dependabot (GitHub) or Renovate automatically create PRs to update dependencies. OWASP

Dependency Check scans dependencies for known CVEs. Outdated dependencies are tech debt plus a security risk. Automate their updates.

What next

Thank you for reading “Technical Debt: How a Startup Kills Itself with Its Own Code”.

Updates to this book, EPUB and PDF files and every guide in the series are on the book page: guides.mmix.biz/en/tech-debt

Maksym Stepanenko, Innova Publishing.